LCC-Graph: A High-Performance Graph-Processing Framework with Low Communication Costs

Yongli Cheng Fang Wang Hong Jiang[‡] Yu Hua Dan Feng Xiuneng Wang
Wuhan National Lab for Optoelectronics, Wuhan, China
School of Computer, Huazhong University of Science and Technology, Wuhan, China
[‡]Department of Computer Science & Engineering, University of Texas at Arlington, USA
{chengyongli, wangfang, csyhua, dfeng, xiunengwang}@hust.edu.cn hong.jiang@uta.edu
Corresponding Author: Fang Wang

***Abstract*—With the rapid growth of data, communication overhead has become an important concern in applications of data centers and cloud computing. However, existing distributed graph-processing frameworks routinely suffer from high communication costs, leading to very long waiting times experienced by users for the graph-computing results. In order to address this problem, we propose a new computation model with low communication costs, called LCC-BSP. We use this model to design and implement a high-performance distributed graph-processing framework called LCC-Graph. This framework eliminates the high communication costs in existing distributed graph-processing frameworks. Moreover, LCC-Graph also minimizes the computation workload of each vertex, significantly reducing the computation time for each superstep. Evaluation of LCC-Graph on a 32-node cluster, driven by real-world graph datasets, shows that it significantly outperforms existing distributed graph-processing frameworks in terms of runtime, particularly when the system is supported by a high-bandwidth network. For example, LCC-Graph achieves an order of magnitude performance improvement over GPS and GraphLab.**

I. INTRODUCTION

Due to the increasing need to analyze, process and mine large real-world graphs, such as social networks, web graphs, chemical compounds and biological structures, there has been a recent surge of interest in distributed graph-processing frameworks in both academia and industry. Several Bulk Synchronous Parallel (BSP) computation model [1] based distributed graph-processing frameworks, such as Pregel [2], GPS [3], Giraph [4], GoldenOrb [5] and Grace [6], have been proposed to handle large-scale graphs. The BSP computation model employs a "think like a vertex" programming model to support iterative graph computation, which considers a graph-computing job as a set of iterations called supersteps. Within each iteration, vertices run in parallel across a distributed cluster of compute nodes. During the execution process of each iteration, vertices interact with each other by passing messages directly, generating an enormously large number of messages. This fine-grained message-passing communication model is inefficient due to the average overhead of each message [2]–[4], [7]–[9].

To improve the communication efficiency, several BSP-based distributed graph-processing frameworks, such as Pregel [2] and GPS [3], leverage the *message buffering* technique to amortize the average overhead of each message. This technique seems to be an effective solution to the problem of inefficient communication. However, even with this *message buffering* technique, BSP-based distributed graph-processing frameworks still suffer from high communication costs. For example, our experiments show that when GPS performs the PageRank [10] algorithm with the *message buffering* technique on a 40Gbps network, the communication cost is responsible for 95% of the overall run time and only 0.9% of the network bandwidth is utilized. The communication cost is defined as the time for vertices to interact with each other, including the communication time for sending messages through the network and the extra sender-side and receiver-side communication overheads. The high communication costs stem mainly from the following four factors.

1) *The bulk of the extra communication volume* that comes from the need to carry the destination vertex name on each message. For most graph algorithms, such as Label Propagation [11], Single-Source Shortest-Paths [7] and PageRank [10], the extra communication volume due to vertex names is responsible for up to 67% of the overall communication volume, as discussed in Section II.
2) *Data copying overhead*. At the sender side, any message generated by a work thread is first sent to the message buffers [2], [3]. When a message buffer is filled up, the message batch in the message buffer is sent to the network. At the receiver side, when a message batch is received by the *message parser*, it parses the message batch and enqueues the messages in the message batch to the message queues of the destination vertices according to the name of each destination vertex [2], [3]. Thus, each vertex can identify the messages sent to itself. There are two rounds of data copying, one at the sender side and the other at the receiver side.
3) *The parsing overhead* that is used by the *message parser* to parse the received message batches [2], [3].
4) *The poor communication bandwidth utilization* that stems from the slow process of message generation and the general-purpose communication protocols that only provide a limited bandwidth.

At the other end of the spectrum of the graph-processing frameworks research, GraphChi [8] has been proposed to process graphs with billions of edges on just one commodity computer, by relying on secondary storage. In each iteration, GraphChi executes subgraphs sequentially. Each execution

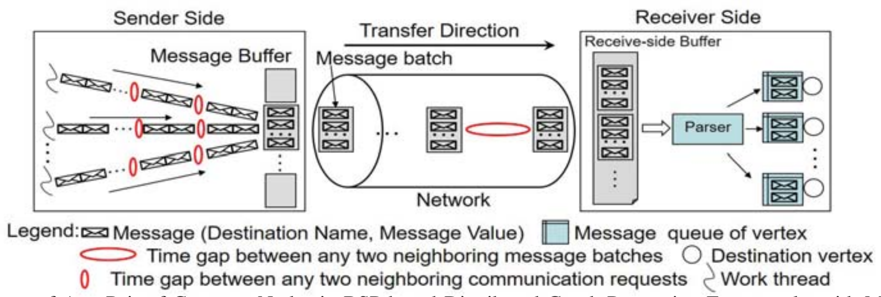

Sender Side
Message Buffer
Transfer Direction
Message batch
Network
Receiver Side
Receive-side Buffer
Parser
Legend: Message (Destination Name, Message Value)
Message queue of vertex
Time gap between any two neighboring message batches
Destination vertex
Time gap between any two neighboring communication requests
Work thread

Fig. 1. The Execution Process of Any Pair of Compute Nodes in BSP-based Distributed Graph-Processing Frameworks with Message Buffering Technique.

process of a subgraph consists of three stages. (1) loading subgraph from disk. (2) computation. (3) saving results to disk. By using the Parallel Sliding Window (PSW), GraphChi has four salient advantages. First, it only uses edge values to implement the interactions among vertices, minimizing the disk I/O workloads. Second, it avoids costly data copying and parsing overheads. Third, it alleviates random accesses to disks to improve the I/O performance. Finally, the computation time of each vertex is reduced to the time for reading/writing its in-edge/out-edge values only.

However, GraphChi is a single-node disk-based graph-processing framework that suffers from poor performance due to its limited scalability and costly disk I/Os. Furthermore, the PSW of GraphChi is not suitable for and cannot be used in distributed graph-processing frameworks because the shards [8] are tightly coupled by PSW, that is, when loading a subgraph S, GraphChi needs to obtain not only in-edges and local out-edges from shard S, but also other out-edges from other shards [8]. Each shard is a disk file that stores all the edges along with their values that have their destination vertices in a given subgraph.

In this paper, inspired by the salient advantages of GraphChi, we propose a high-performance distributed in-memory graph-processing framework, called LCC-Graph. The high performance of LCC-Graph stems from the low communication costs and reduced computation time. LCC-Graph is significantly different from GraphChi because the former is designed to reduce the high communication costs experienced by existing distributed graph-processing frameworks while the latter aims to improve the performance of single-node disk-based graph-processing frameworks by reducing disk I/O costs. By designing and implementing LCC-Graph, this paper makes the following three contributions.

1) *A computation model with low communication costs*, called LCC-BSP, that decomposes each superstep into two distinct steps of computation and communication. In the computation step, each vertex does computation task by reading and writing its edge values directly, reducing the computation time. In the communication step, each compute node exchanges full remote out-edge data blocks with other compute nodes to implement edge-value transfers among inter-node vertices. The clear advantage of this decomposition of superstep is that the interactions among vertices will be finished instantaneously and simultaneously in a well-orchestrated concurrent manner after the computation step that runs for only a short period of time.
2) *An edge-block based data representation.* By organizing the edge values of each compute node intelligently in the preprocessing phase, this data representation provides four salient advantages that enable the reduced communication costs and computation time in LCC-BSP. First, it eliminates the high extra volume of communication in existing BSP-based distributed graph-processing frameworks. Second, it avoids the data copying and batch parsing overheads. Third, it enables the network bandwidth capacity to be efficiently utilized. Finally, the computation workload of each vertex is reduced.
3) *The design and prototype implementation of LCC-Graph.* LCC-Graph is also capable of improving the performance of those graph algorithms with relatively small numbers of interactions among vertices, despite the fact that, in most graph algorithm jobs [10], [12], [13], almost all the vertices interact with each other. Moreover, it also flexibly supports some peculiar requirements of graph algorithms even if these cases rarely happen. These requirements include messages sent to non-neighbor vertices and multiple messages sent to a single destination vertex.

The rest of the paper is structured as follows. Background and motivation are presented in Section II. Section III introduces our LCC-BSP, a high-performance computation model. Section IV presents LCC-Graph, our LCC-BSP based graph-processing framework. Experimental evaluations of the LCC-Graph prototype are presented in Section V. We discuss related work in Section VI and conclude the paper in Section VII.

II. BACKGROUND AND MOTIVATION

In this section, we present a brief introduction to the Bulk Synchronous Parallel (BSP) computation model, in order to explore the efficiency issues of existing BSP-based distributed graph-processing frameworks. We then introduce GraphChi, a single-node disk-based graph-processing framework. GraphChi has four salient advantages that reduce the disk I/O costs. The insights gained through the efficiency issues of existing BSP-based distributed graph-processing frameworks and the advantages of GraphChi help motivate us to propose an LCC-BSP computation model that eliminates the high communication costs incurred by existing BSP-based distributed graph-processing frameworks, and reduces the computation workload of each vertex.

A. *BSP Computation Model*

BSP-based distributed graph-processing frameworks, such as Pregel [2], GPS [3], Giraph [4] and GoldenOrb [5], can easily express graph algorithms in a fully vertex-centric fashion. In these frameworks, the vertices of a directed graph are distributed across compute nodes during the preprocessing phase. A graph-computing job consists of iterations called supersteps, which terminates when all vertices vote to stop computation. In each superstep, a user-defined **Compute(v)** function is invoked for each vertex **v**, conceptually in parallel. In fact, due to the limited core count of each compute node, the executions of vertices are assigned to a limited number of work threads that execute concurrently. All work threads of compute nodes start simultaneously. Then, each work thread loops through its assigned vertices by using the vertex-program (i.e., the **Compute(v)** function). Each vertex-program first does a computation task and then a communication task by sending messages to its neighbors. The superstep ends when the last communication task has finished.

As mentioned before, the fine-grained and intermittent nature in which the very large number of messages are generated and exchanged in each superstep, by the vertices executed by the work threads, results in very high communication costs [8], [14], [15]. While several BSP-based distributed graph-processing frameworks, such as Pregel [2] and GPS [3], address this problem by leveraging the *message buffering* technique to amortize the average overhead of each message, they continue to suffer from the high communication costs [4], [7], [8], even when the *message buffering* technique is employed. Figure 1 shows the execution process of any pair of compute nodes in the BSP-based distributed graph-processing frameworks with the *message buffering* technique. The main reasons for the high communication costs are fourfold.

First, the bulk of the extra communication volume is attributed to the need to carry the destination vertex name on each message. The names of the destination vertices are used by the receiver sides to route messages to the message queue of their respective destination vertices [2], [3]. For most graph algorithms, such as Label Propagation [11], Single-Source Shortest-Paths [7] and PageRank [10], a message usually carries a value with a short size, such as a 4-byte integer or floating-point number used to store the label value, shortest-path value or pagerank value. However, the size of the name of each destination vertex is usually longer than that of the message value (i.e., payload). For example, in order to handle the graphs with more than four billion vertices, graph-processing frameworks must use an 8-byte long-integer number to identify each vertex. In this case, the extra communication volume due to vertex names is responsible for 67% of the overall communication volume.

Second, as discussed in Section I, there are two rounds of data copying, one at the sender side and the other at the receiver side, i.e., sending the messages to the message buffers and enqueuing the messages in the message batches to the message queues of the destination vertices [2], [3].

Third, at the receiver side, in order to dispatch the messages in the message batches to the message queues of the destinations vertices, the *message parser* parses the message batches, resulting in the high parsing overhead [3].

Finally, the network bandwidth capacity is utilized poorly. The reasons are twofold. First, the message buffers are filled up rather slowly because any two consecutive buffer-filling operations of a work thread are separated by the three distinctive tasks of computation, message generation and decision on the placement of each message. Therefore, it will take a relatively long time to fill up a message buffer of a reasonably large size to achieve a good amortization, which causes a long idle period of the network. In fact, the larger the size of the message buffers is, the longer the idle periods of the network are. Hence, the size of the message buffer is a tradeoff between the gains from the batched communication and the idle periods of the network. Second, existing distributed graph-processing frameworks are usually designed on the basis of the general-purpose communication protocols that only provide a limited bandwidth. This limit hinders these frameworks from speeding up the transfers of the batched messages, particularly when the system is supported by a high-bandwidth network.

B. *GraphChi*

Graphchi [8] is a single-node disk-based graph processing framework. It introduces a novel mechanism called Parallel Sliding Windows (PSW) to reduce the disk I/O costs to improve performance. GraphChi works as follows. In preprocessing stage, the vertices of the input graph are divided into continuous but disjoint ***P*** intervals, each of which is associated with a shard. Each **shard** is a disk file that stores all the edges along with their values that have their destination vertex labels in a given interval. Edges are stored in order of their *sources*. This data representation has a clear advantage, that is, for any loaded subgraph ***S***, shard ***S*** contains the information of all the in-edges and local out-edges of the loaded subgraph, and all the out-edges with their destination vertices in a given unloaded subgraph ***R*** can be obtained from a contiguous disk file chunk in the shard ***R***. In each iteration, GraphChi executes ***P*** subgraphs sequentially. Each execution process of a subgraph ***S*** consists of three stages. (1) loading subgraph from disk. GraphChi obtains all the in-edges and local out-edges by reading the full shard ***S*** and other out-edges by reading ***P***-1 contiguous disk file chunks in other ***P***-1 shards. (2) executing the user-defined **Update(v)** function for each vertex **v** in the loaded subgraph. (3) saving results to disk.

There are four salient advantages of PSW. First, it minimizes the disk I/O workloads. In the stage of saving results, GraphChi only needs to write the edge values back to the disk files (shards) since only the edge values are updated during computation [8]. However, in existing distributed graph-processing frameworks, each message consists of a destination name and a message value. Second, it avoids costly data copying and parsing overheads. The edge values indexed by the vertices are available immediately for the vertices after the subgraph is loaded. Also, the saving results stage can be executed immediately after the computation stage is finished. Third, by reading/writing the full shard and the contiguous disk file chunks, it alleviates random accesses to improve the I/O performance. Finally, the computation time of each vertex is reduced to the time for reading/writing its in-edge/out-edge values only, eliminating the time for message generation, decision on the placement of and filling each message in existing distributed graph-processing frameworks.

However, as mentioned in Section I, the PSW of GraphChi is not suitable for and cannot be used in distributed graph-processing frameworks that are of higher scalability and performance than GraphChi.

III. LCC-BSP Computation Model

LCC-BSP, like the BSP computation model, considers a graph-computing job as a set of supersteps. However, unlike BSP, the communication process in LCC-BSP is decoupled from its computation process by explicitly dividing each superstep into a computation step and a communication step. At the end of each superstep, a barrier is required to ensure the determinism of graph-computing jobs.

In the computation step, a user-defined **Update(v)** function is invoked for each vertex **v** in parallel. Inside **Update(v)**, the vertex **v** updates its state by its in-edge values and then updates its out-edge values. The in-edge values of vertex **v** were updated by the source vertices of the in-edges in the previous superstep, and the out-edge values of vertex **v** will be used by the destination vertices of the out-edges in the next superstep.

In the communication step, edge values are moved to implement the interactions among vertices, since the out-edges of a vertex are the in-edges of its neighboring vertices. For each compute node, all out-edge values for a given remote compute node are organized into a single full remote out-edge data block in the preprocessing phase judiciously. Similarly, all in-edge values for a given remote compute node are organized into a single full remote in-edge data block. A remote out-edge data block is an exact copy of a remote in-edge data block of a remote compute node. After the computation step has finished, each compute node only needs to send *P*-1 remote out-edge data blocks to *P*-1 remote compute nodes simultaneously, to update their respective copies, where *P* is the number of compute nodes.

A synchronous barrier is added between the computation step and the communication step in each superstep. This barrier ensures the determinism, i.e., all out-edge values are updated completely before sending the out-edge value blocks to remote compute nodes. Intuitively, this barrier can potentially cause some synchronization cost in each computation step. However, our experimental observations show that this barrier does not delay the overall run time of each superstep. First, the time spent on the computation step of each compute node is very short. Second, the slowest compute node in the computation step is usually also the slowest one in the communication step.

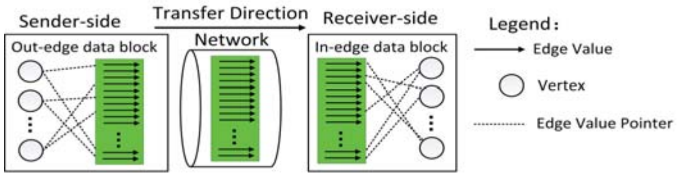
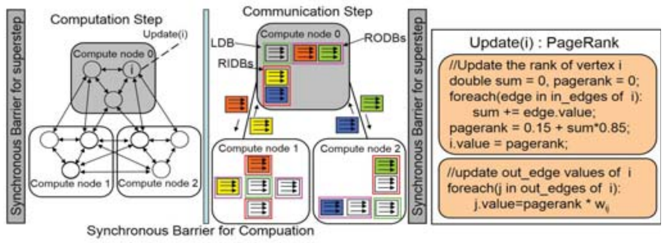


Fig. 2. The Execution Process of Any Pair of Compute Nodes in the LCC-BSP Computation Model.



```
//Update the rank of vertex i
double sum = 0, pagerank = 0;
foreach(edge in in_edges of  i):
    sum += edge.value;
pagerank = 0.15 + sum*0.85;
i.value = pagerank;
```

```
//update out_edge values of  i
foreach(j in out_edges of  i):
    j.value=pagerank * w_ij
```

Fig. 3. An Example of LCC-BSP Computation Model.

A. Performance Analysis

The clear advantage of this computation model is that, in each superstep, the interactions among vertices will be finished instantaneously and simultaneously in a well-orchestrated concurrent manner after the computation step that runs only a short period of time. Furthermore, the key advantage of the communication step is that each out-edge data block is an exact copy of an in-edge data block of a remote compute node, that is, each edge value has an identical and fixed position in the two copies. Using the fixed position, each edge value indexed by its vertex can be identified by the vertex directly in both the sender side and the receiver side without the vertex name, avoiding the high data copying and parsing overheads. As shown in Figure 2, there are four salient features that contribute to the high performance of this computation model.

The reduced communication volume. LCC-BSP eliminates high extra volume of communication required to carry the name of destination vertex on each message in existing BSP-based distributed graph-processing frameworks. Since the out-edge data blocks that are sent to the network only include the edge values.

The eliminated data copying and parsing overheads. As mentioned before, the **Update(v)** function reads/writes the edge values directly in both sender side and the receiver side, avoiding the high data copying and parsing overheads.

The reduced computation time. In each computation step, the computation time is much shorter than that of the existing BSP-based distributed graph-processing frameworks, since the computation time of each vertex is reduced to the time for reading/writing its in-edge/out-edge values only, eliminating the time for message generation, decision on the placement of and filling each message in the latter.

Highly efficient communication. First, in each superstep, LCC-Graph only causes a very short idle period of the network occupied by the computation step due to the reduced computation time. Second, in the communication step, each compute node only needs to send *P*-1 remote out-edge data blocks to *P*-1 remote compute nodes simultaneously. This well-orchestrated concurrent manner provides a sufficiently large instantaneous network workload that enables the network bandwidth capacity to be efficiently utilized, particularly when the system is supported by a high-bandwidth network that is more easily accessible than ever before. Intuitively existing BSP-based distributed graph-processing frameworks with message buffers can also provide a large instantaneous network workload by using large enough message buffers. However, larger message buffers cause longer idle periods of the network used by the work threads to fill up the message buffers due to the slow process of message generation, as discussed in Section II. Moreover, the optimization of the network ecosystem presented in Section IV helps further speed up the transfer of the out-edge data blocks. In the context of this paper, the network ecosystem is interpreted as the combination of the network hardware and communication protocol software.

B. *High Flexibility*

LCC-BSP also flexibly supports some peculiar requirements of graph algorithms by using a *mignon message block* attached to each out-edge data block, even if these cases rarely occur. These requirements include messages sent to non-neighbor vertices and multiple messages sent to a single destination vertex. Like existing BSP-based distributed graph-processing frameworks, these messages carried by the *mignon message blocks* are separated and routed to their destination vertices by the receiver side. They also include the names of the destination vertices. However, the introduced extra overhead is negligible since the number of these messages is very small and the overhead associated with each of these messages is amortized by the efficient communication of LCC-BSP. This feature of flexibility is also provided by Pregel [2]. By using this compensatory technique, LCC-BSP computation model acts exactly as the BSP computation model.

C. *An Example*

To better illustrate the LCC-BSP computation model we use an example of the computation of PageRank [3]. In this example, the directed graph is organized into three subgraphs residing in three compute nodes. Each subgraph includes a local edge data block (**LDB**), two remote in-edge data blocks (**RIDBs**) and two remote out-edge data blocks (**RODBs**). Figure 3 shows a superstep of LCC-BSP computation model processing the PageRank algorithm.

In the computation step, each vertex **v** is executed by using the **Update(v):PageRank** function concurrently with other vertices. Consider vertex **i** as an example. First, a new *pagerank* is calculated according to the in-edge values of vertex **i**. Then the value of vertex **i** is updated with the new *pagerank*. Finally the out-edges' values of vertex **i** are updated based on new *pagerank*. The edge values, organized into edge data blocks, are read and written by the **Update(v):PageRank** function directly, avoiding data copyings. In the communication step, two remote out-edge data blocks of each compute node are sent to two other compute nodes, to update their respective copies. In this example, compute node 0 sends the red block to compute node 1 and the green block to compute node 2. Meanwhile each compute node will receive two remote out-edge data blocks from two other compute nodes to update its corresponding remote in-edge data blocks. In this example, compute node 0 receives the yellow block from compute node 1 and the blue block from compute node 2.

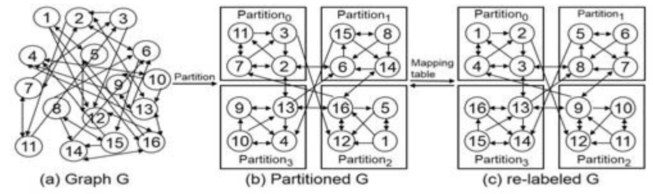


Fig. 4. An Example of Graph Partitioning and Re-labeling Vertices. Circles denote vertices with their labels. Note that each partition is a subset of vertices in graph G. To help us better understand the subgraph construction process, the edges are also illustrated in subfigures (b) and (c).

D. *Challenges*

There are two key challenges in the implementation of the LCC-BSP computation model. The first one is to organize all the out-edge/in-edge values from a given compute node to another into a single remote out-edge/in-edge data block. The other is to guarantee that each out-edge data block is an exact copy of an in-edge data block of a remote compute node. To address these challenges, the new *edge-block based data representation* is introduced, as presented in Section IV-A.

IV. LCC-GRAPH FRAMEWORK

In this section, we present LCC-Graph. Key components and unique features of the LCC-Graph are detailed in various subsections next.

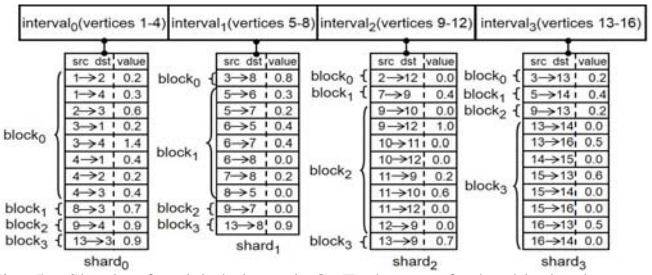
A. *The Edge-block Based Data Representation*

In this subsection, we present the edge-block based data representation that provides four salient features of the LCC-BSP computation model. We will now describe how the input graph is partitioned and organized into edge-block based subgraphs during the preprocessing phase. Each subgraph resides in the memory of a compute node.

Definitions: Let G =(V,E) denote an input graph with its vertex set V and edge set E, and let $\text{Partition}_0 \cup \text{Partition}_1 \cup \cdots \cup \text{Partition}_{P-1}$ = V be the *P* partitions of V, where $\text{Partition}_i \cap \text{Partition}_j = \emptyset$ $(i \neq j)$. For each Partition_i, the vertices in Partition_i, along with their edges, are defined as a subgraph of the input graph. The input is a directed graph and an undirected graph can be treated as a directed one by considering each undirected edge as two opposite directed edges.

Graph Partitioning: By using a user-specified graph partitioning method, the input graph is divided into *P* partitions, as depicted in Figure 4(b), where *P* is the number of compute nodes.

Relabeling Vertices: It is assumed that the vertices are labeled from 1 to |V|. After dividing the input graph into *P* partitions, the vertices of the graph are unordered again. We re-label vertices of the *P* partitions sequentially to make the labels of vertices form *P* continuous but disjoint intervals, corresponding to the *P* partitions. Consider as an example the re-labeled graph G, shown in Figure 4(c), the labels of vertices consist of four continuous but disjoint intervals, i.e., interval_0:


interval$_0$(vertices 1-4)
interval$_1$(vertices 5-8)
interval$_2$(vertices 9-12)
interval$_3$(vertices 13-16)
shard$_0$
shard$_1$
shard$_2$
shard$_3$

shard$_0$

	src dst	value
block$_0$	1→2	0.2
	1→4	0.3
	2→3	0.6
	3→1	0.2
	3→4	1.4
	4→1	0.4
	4→2	0.2
	4→3	0.4
block$_1$	8→3	0.7
block$_2$	9→4	0.9
block$_3$	13→3	0.9

shard$_1$

	src dst	value
block$_0$	3→8	0.8
block$_1$	5→6	0.3
	5→7	0.2
	6→5	0.4
	6→7	0.4
	6→8	0.0
	7→8	0.2
	8→5	0.0
block$_2$	9→7	0.0
block$_3$	13→8	0.9

shard$_2$

	src dst	value
block$_0$	2→12	0.0
block$_1$	7→9	0.4
block$_2$	9→10	0.0
	9→12	1.0
	10→11	0.0
	10→12	0.0
	11→9	0.2
	11→10	0.6
	11→12	0.0
	12→9	0.0
block$_3$	13→9	0.7

shard$_3$

	src dst	value
block$_0$	3→13	0.2
block$_1$	5→14	0.4
block$_2$	9→13	0.2
block$_3$	13→14	0.0
	13→16	0.5
	14→15	0.0
	15→13	0.6
	15→14	0.0
	15→16	0.0
	16→13	0.5
	16→14	0.0

Fig. 5. Shards of re-labeled graph G. Each row of edge blocks denotes a directed edge with its value.

[1-4], interval$_1$: [5-8], interval$_2$: [9-12] and interval$_3$: [13-16], corresponding to Partition$_0$, Partition$_1$, Partition$_2$ and Partition$_3$ respectively. By using the continuous but disjoint intervals, it is easy to identify which partition the vertices belong to, instead of accessing the global graph-partitioning information. This measure further enables the judicious date representation, described next.

Shards: A **shard** is a consecutive memory chunk that stores all the edges along with their values that have their destination vertex labels in a given interval. That is, a shard is associated with each interval. Edges are stored in order of their $sources$. Shards are labeled from 0 to P-1. As shown in Figure 5, four shards are generated for the four intervals, labeled as shard$_0$, shard$_1$, shard$_2$ and shard$_3$ respectively.

Constructing Subgraphs: In order to create a subgraph$_p$ for the vertices in Partition$_p$, where $0 \leq p \leq P$-1 and P is the number of compute nodes, in-edges and out-edges of these vertices need to be obtained. First, shard$_p$ contains the information of all the in-edges and local out-edges (their $destinations$ are also in Partition$_p$) for the vertices in Partition$_p$, so the in-edges and local out-edges can be easily obtained from shard$_p$. Consider the re-labeled Graph G, shown in Figure 4(c), vertex "2" residing in Partition$_0$ has two in-edges ("1→2" and "4→2") and a local out-edge ("2→3"), which are stored in the first row, seventh row and the third row in the shard$_0$ respectively, as shown in Figure 5. Second, since the out-edges of a given vertex are the in-edges of its neighboring vertices, the remote out-edges of the vertex are stored in the other P-1 shards. Consider the vertex "2", it has a remote out-edge("2→12") which is stored in the first row in shard$_2$. Moreover, since all the labels of the vertices in Partition$_p$ are within the interval$_p$ and the edges in each shard are stored in order of their $sources$, the remote out-edges of the vertices in Partition$_p$ with their $destinations$ within a given remote Partition$_j$ are stored in a contiguous memory chunk in the shard$_j$. Hence, the full remote out-edges of the vertices in Partition$_p$ can be obtained from the P-1 contiguous memory chunks in the other P-1 shards. However, in a distributed setting, it is inefficient to deploy P shards on all compute nodes of the cluster. Thus, each shard is decomposed into P edge blocks (labeled 0 to P-1) according to the interval that contains the source vertex labels corresponding to the edges. As shown in Figure 5, each shard is split into four edge blocks: block$_0$, block$_1$, block$_2$ and block$_3$. There are P^2 edge blocks in total, and we use B(x, y) to identify an edge block, with block label x and shard label y. Consider Figure 5 as an example, in order to construct subgraph$_0$, all the in-edges and local out-edges of vertices in Partition$_0$ can be obtained first by a full reading of the shard$_0$ (from block$_0$ to block$_3$).Then all the remote out-edges can be obtained from the block$_0$ of shard$_1$, block$_0$ of shard$_2$ and block$_0$ of shard$_3$ respectively.

After blocking, we can construct P subgraphs for P compute nodes. Each subgraph$_p$ consists of a local-edge block (**LB**) B(block$_p$, shard$_p$), P-1 remote in-edge blocks (**RIBs**) B(block$_i$, shard$_p$) and P-1 remote out-edge blocks (**ROBs**) B(block$_p$, shard$_j$), where $0 \leq i \leq P$-1, $0 \leq j \leq P$-1, $i \neq p$, $j \neq p$. Consider the re-labeled graph G in Figure 4(c) as an example, four subgraphs are constructed, as shown in Figure 6. Each subgraph$_p$ ($0 \leq p \leq 3$) consists of a local-edge block, three remote in-edge blocks and three remote out-edge blocks. For each subgraph$_p$, the local-edge block and the three remote in-edge blocks are in the column p, depicted using the same color in the figure; the three remote out-edge blocks, depicted using different colors, are in the row p. Each remote out-edge block includes all the edges whose destination vertices reside within a given remote partition. Each remote in-edge block includes all the edges whose source vertices reside within a given remote partition. The local-edge block B(block$_p$, shard$_p$) is a special case because both the source vertices and destination vertices of the edges are in the partition$_p$.

B. Edge Data Block based Communication

Block dependencies: For each subgraph$_p$ ($0 \leq p \leq P$-1), each remote out-edge block B(block$_p$, shard$_j$) is a copy of the remote in-edge block B(block$_p$, shard$_j$) of subgraph j, where $0 \leq j \leq P$-1 and $p \neq j$. As shown in Figure 6, consider subgraph$_0$, the remote out-edge block B(block$_0$, shard$_1$) of subgraph$_0$ is a copy of the remote in-edge block B(block$_0$, shard$_1$) of subgraph$_1$; the remote out-edge block B(block$_0$, shard$_2$) of subgraph$_0$ is a copy of the remote in-edge block B(block$_0$, shard$_2$) of subgraph$_2$; and the remote out-edge block B(block$_0$, shard$_3$) of subgraph$_0$ is a copy of the remote in-edge block B(block$_0$, shard$_3$) of subgraph$_3$.

Communication: Since only the edge values are mutated during computation, each edge block B(x, y) is split into an adjacency block B_adj(x, y) and an edge data block B_data(x, y). The adjacency blocks store the topological structure of the input graph and the edge data blocks store the edge values. In the communication step, the P-1 remote out-edge data blocks of each compute node will be sent to the other P-1 compute nodes to update their respective copies.

Local edge data block B_data(p, p): The local edge block B(p, p) of each subgraph$_p$ is a special case since both the source and destination vertices of its edges belong to partition$_p$. In the computation step, conflicts can occur when the values of these edges are accessed by the source vertices and destination vertices of these edges simultaneously, which breaks determinism. To guarantee determinism, we implement two copies of the local-edge data block B_data(p, p). One is used for reading and the other for writing. In the next superstep, the two copies switch their roles.

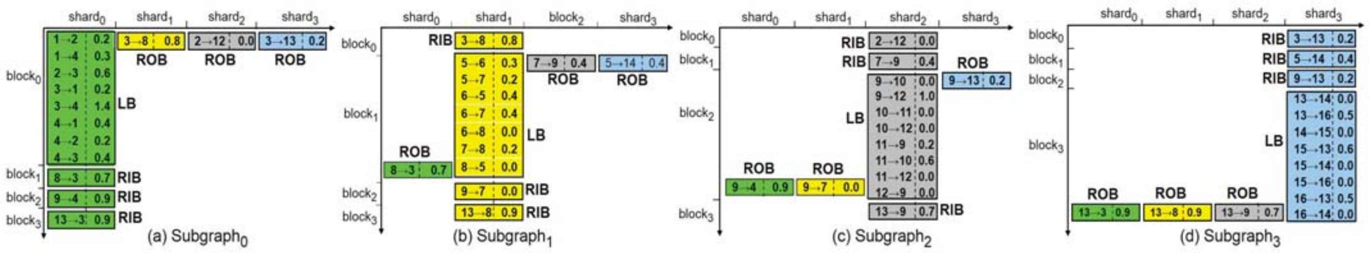


Fig. 6. Subgraphs of G. The x axis denotes the shard labels, and the y axis denotes the block labels corresponding Figure 5.

C. Network Ecosystem

In the network ecosystem, the TCP/IP socket interface calls were initially used for inter-node communication and LCC-Graph was performing over a 1Gbps Ethernet, which can only provide $\sim$110MB/s of actual network bandwidth. Using the edge data block based communication model, LCC-Graph consumes the whole 1Gbps Ethernet network bandwidth, i.e., it can obtain better performance if running on a higher bandwidth network. Hence, we run LCC-Graph on a 40Gbps Infiniband network. However, the TCP/IP protocol now becomes the new bottleneck, due to its low efficiency for multi-layer complex structures. In fact, LCC-Graph can only obtain $\sim$1.2GB/s actual application bandwidth on TCP/IP over the 40Gbps Infiniband network, far below the peak performance. Hence a more efficient communication protocol is required.

The InfiniBand Architecture (IBA) [16] defines a switched network fabric for the interconnection of processing nodes, which provides a communication and management infrastructure for inter-processor communication. In an InfiniBand network, processing nodes are connected to the fabric by channel adapters (CAs). The InfiniBand Architecture supports two semantics, channel and memory. In memory semantics, InfiniBand supports remote direct memory access (RDMA) operations that include RDMA write and RDMA read. RDMA operations are single-sided and do not incur software overheads on the remote side, resulting in efficient communication. Hence, we have implemented LCC-Graph on the InfiniBand Architecture, using RDMA communication operations. In this case, LCC-Graph can obtain $\sim$2.53GB/s actual application bandwidth, further shortening the communication time.

D. Remote Out-edge Data Block Compression

For most graph algorithms, almost all the edge values are updated in supersteps, such as PargeRank [10], Community Detection [13] and Connected Components [12]. In order to further speed up the graph algorithms with only a few updated edge values, a remote out-edge data block compression (CoDB) scheme is introduced in LCC-Graph.

For each compute node, we define a CoDB for each out-edge data block. Each element in the CoDB includes an *offset* and an edge value. The *offset* indicates the edge value offset in the out-edge data block. The edge value is the new value of the edge that is updated in this superstep. Each *offset* is a 4-byte integer number which is large enough to store the maximum value of the offsets due to the limited number of the edge values in each out-edge data block. The extra communication volume for carrying the *offsets* is smaller than that of existing BSP-based distributed graph-processing frameworks since the size of destination vertex names on the messages in the latter is larger than the size of the *offsets*. Like the latter, this scheme also leads to extra overhead at the receiver side. However, by using this scheme, the communication cost of LCC-Graph is still significantly smaller than that of the latter, due to the smaller extra communication volume and higher communication efficiency.

Update Counters (UCs): LCC-Graph defines a UC for each remote out-edge data block to record the number of updated edge values in the remote out-edge data block. In the computation step, when an edge is updated, the UC is increased by 1 and a CoDB element will be added to the CoDB. We call this process *compression* that is enabled by a configurable option. Users can disable this option by default for higher performance. Because, for most graph algorithms, almost all the edge values are updated in supersteps.

Threshold Values (TVs): In the communication step, if the remote out-edge data block compression option is enabled, the CoDBs will be sent to other compute nodes, instead of the full remote out-edge data blocks. However, CoDB introduces additional communication volume to carry the *offset* for each edge value. Hence, a TV is required for each remote out-edge data block to control the ratio of compressed edge values, where $0 \leq TV \leq 1$. In the compression process, if the ratio of compressed edge values to total edge values of the remote out-edge data block reaches the TV, compression is abandoned. In this case, the remote out-edge data block will be sent in the communication step.

When a compute node has received a CoDB, it uses the *offset* of each element to locate the address of the edge value in the corresponding remote in-edge data block and replace the edge value using the new one.

V. EXPERIMENTAL EVALUATION

In this section, we conduct extensive experiments to evaluate the performance of LCC-Graph. Experiments are conducted on a 32-node cluster. Each compute node has two quad-core Intel Xeon E5620 processors with 24GB of RAM. Nodes are connected via a 40Gbps InfiniBand network and a 1Gpbs Ethernet for high-bandwidth and low-bandwidth interconnect evaluations respectively.

Graph Algorithms: We implement several graph algorithms to evaluate LCC-Graph: Single-Source Shortest-Paths (SSSP) [7], PageRank (PR) [10], Community Detection (CD) [13] and Connected Components (CC) [12]. The SSSP algorithm computes the distance of the shortest path from a given source vertex *u* to each other vertex in a graph. The PR algorithm is used by Google Search to rank websites in their search engine. The CC algorithm finds connected components of a given graph, i.e., a maximum set of vertices in which any pair of vertices can reach each other. The CD algorithm works as follows. Each vertex is initially assigned a unique label. Each vertex updates its label with the label most frequently used by its neighbors. This process is repeated until a stable set of labels is reached. We define the sets of vertices that have the same labels as "network communities".

Baseline Frameworks: We compare LCC-Graph with two baseline frameworks. One is an up-to-date version of GPS, which is an open-source Pregel implementation from Stanfords InfoLab [3]. It is a representative BSP-based distributed graph-processing framework. The other is GraphLab, an open-source project originated at CMU [17] and now supported by GraphLab Inc. GraphLab is a representative distributed shared-memory graph-processing framework. We use the latest version of GraphLab 2.2 (released in March 2014), which supports distributed computation and incorporates the features and improvements of PowerGraph [18], [19].

Datasets: We evaluate LCC-Graph using several real-world graph datasets that are summarized in Table I. These datasets are used frequently in many published comparative evaluations of graph-processing frameworks [3], [8], [19].

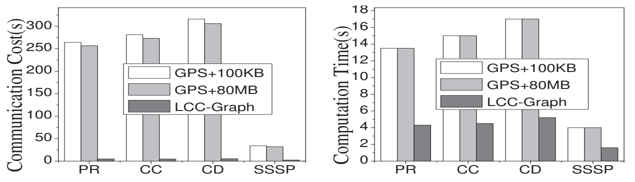
Preprocessing: We conduct experiments to evaluate GPS, GraphLab and LCC-Graph in terms of preprocessing time. The experimental results show that the preprocessing time of LCC-Graph is similar to that of GPS and slightly shorter than that of GraphLab when the same graph partitioning method is used. The preprocessing times are not included in calculations in the other experiments described in the following subsections.

A. *Runtime Breakdown*

Experiments are conducted to investigate LCC-Graph and GPS in terms of communication cost and computation time. GraphLab is excluded from this evaluation since it is difficult to explicitly measure its communication cost due to its distributed shared-memory technique. Each framework runs the PR, CD, CC and SSSP graph algorithms on 24 compute nodes with the Twitter-2010 graph. Compute nodes are connected via a 40Gbps InfiniBand network. PR runs 10 supersteps for each experiment.

Communication Cost: The communication cost is defined as the time spent by vertices to interact with one another. In LCC-Graph, each superstep is explicitly divided into a computation step and a communication step. Hence, in each superstep, the communication cost is the time for the communication step. For GPS, the communication cost consists of the sender-side communication overhead, the time spent on sending message batches, and the receiver-side communication overhead.

For fair comparison, GPS is evaluated with two configurations of message buffer size, i.e., 100KB (the default value) and 80MB which is large enough to accommodate all the messages sent from one compute node to another in



(a) Communication Cost (b) Computation Time

Fig. 7. Runtime Breakdown.

each superstep. We record the communication cost for each experiment. Figure 7(a) shows the difference between LCC-Graph and GPS in terms of communication cost. GPS with the 80MB buffer configuration only gains less than 6% improvements in the communication cost, compared to its default configuration. The reasons are the high extra volume of communication, limited effectiveness of the *message buffering* technique and poor communication bandwidth utilization in GPS, as discussed in Section II. However, the communication cost of LCC-Graph is 59x, 60x, 66x, and 14x shorter than that of GPS with the 80MB buffer configuration when running the PR, CD, CC and SSSP algorithms respectively. The reasons are the reduced communication volume, the highly efficient communication and the elimination of the extra overhead at the receiver side. We also observe that the communication cost gap between LCC-Graph and GPS is narrower for SSSP than for the other graph algorithms. The reason is that the SSSP graph algorithm generates only a few inter-vertex interactions during the execution process. Even so, the communication cost of LCC-Graph is still 14x shorter than that of GPS.

The experimental results indicate that the communication cost of GPS dominates the overall run time in each graph-computing job. For example, when GPS performs the PageRank algorithm, the communication cost is responsible for 95% of the overall run time. On the contrary, it is the low communication cost of LCC-Graph that contributes to its high-performance in each graph-computing job.

Computation Time: In these experiments, we also measure the time spent on computation in each graph-computing job. Experimental results, as shown in Figure 7(b), indicate that the computation time of LCC-Graph is 2.5x-3.4x shorter than that of GPS. As discussed in Section III, this performance improvement stems from the reduced computation work of each vertex. The reduced computation time is another contributor to the high-performance of LCC-Graph.

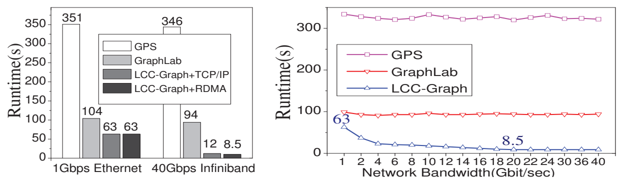
B. *Effects of Network Bandwidth*

Experiments are also conducted to study the effect of the network bandwidth on the performance of these frameworks. In these experiments, each framework is deployed on 16 compute nodes, and runs 10 supersteps of PR on the Twitter-2010 graph.

Over the 1Gbps Ethernet: We first test LCC-Graph against GraphLab and GPS over the 1Gbps Ethernet. As illustrated in Figure 8(a), LCC-Graph is 5.6x and 1.7x faster than GPS and GraphLab respectively. The run-times of LCC-Graph are similar when running on either TCP/IP or RDMA. The reason is that edge data block based communication model saturates

TABLE I
GRAPH DATASETS SUMMARY.

DataSets	$\|V\|$	$\|E\|$	Type	Avg/In/Out degree	Max -/In/Out degree	Largest SCC
LiveJournal [20]	4.8×10^6	69×10^6	Social Network	18/14/14	20K/13.9K/20K	3.8M (79%)
Twitter-2010 [21]	41×10^6	1.4×10^9	Social Network	58/35/35	2.9M/770K/2.9M	33.4M (80.3%)
UK-2007-05 [21]	106×10^6	3.7×10^9	Web	63/35/35	975K/15K/975K	68.5M (64.7%)



(a) Different Networks (b) Network Bandwidth Scalability

Fig. 8. Effects of Network Bandwidth.

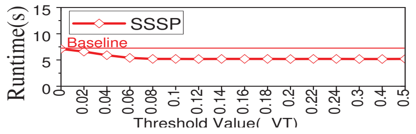


Fig. 9. Effects for CoDB.

the 1Gbps network bandwidth when running on either TCP/IP or RDMA. In fact, the actual obtained bandwidth, calculated as the size of the out-edge data blocks divided by the communication time, is ∼110MB/s, which reaches the upper limit of the 1Gbps Ethernet network. In these experiments, the network bandwidth is the performance bottleneck in LCC-Graph.

Over the 40Gbps Infiniband: We then evaluate LCC-Graph against GraphLab and GPS over the 40Gbps Infiniband. The experimental results shown in Figure 8(a) indicate that LCC-Graph with RDMA is 1.4x faster than LCC-Graph with TCP/IP, due to higher efficiency of RDMA, and is 41x and 11x faster than GPS and GraphLab respectively. Compared with the 1Gbps Ethernet, LCC-Graph, when running over the 40Gbps Infiniband, obtains significant performance improvements, which is not the case for GPS and GraphLab that fail to observe any significant performance improvements. The experimental results indicate that LCC-Graph has significantly higher efficiency when the system is supported by a high-quality network ecosystem.

Scalability: We conduct experiments to evaluate the scalability of these frameworks in terms of network bandwidth. Each experiment is repeated by gradually increasing the network bandwidth from 1Gbps to 40Gbps. As illustrated in Figure 8(b), LCC-Graph, running on RDMA, achieves the peak performance when the network bandwidth is limited to 20Gbps. Consider the 1Gbps-based configuration as the baseline, the peak performance (8.5s) is 7.4x higher than the baseline performance (63s). When LCC-Graph achieves its peak performance, the measured actual obtained bandwidth is ∼2.53GB/s. However, a higher network bandwidth does not contribute to higher performances of GraphLab and GPS.

C. Out-edge Data Block Compression (CoDB)

We also study the CoDB that can help speedup the graph-computing jobs with a few inter-vertex interations. LCC-Graph runs SSSP on 24 compute nodes with the Twitter-2010 graph, ranging the TV (threshold value) from 0 to 0.5. In fact, CoDB is disabled when TV=0. We regard this case as the baseline. As shown in Figure 9, the runtime decreases gradually and reaches a minimum value when TV=0.08. The reason is that a larger TV value enables more out-edge data blocks to be compressed, gaining a reduction in communication time. The runtimes maintain the minimum value when TV ranges from 0.08 to 0.5. The reason is that the maximally connected component of the Twitter graph only has ∼3 million vertices while the Twitter graph has ∼41 million vertices. The ratio of the scheduled vertices is less than 7.2%, leading to a small proportion of edges being updated during the execution process. Hence, most out-edge data blocks can be compressed when TV is larger than 0.08. Overall, CoDB is able to reduce the runtime of SSSP by 26%.

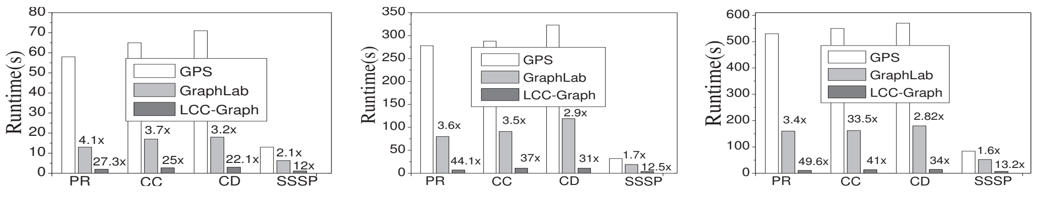
D. Comprehensive Evaluation

In order to demonstrate the superior performance of the LCC-Graph framework to the baseline frameworks, we evaluate LCC-Graph comprehensively with different graph algorithms on various graph datasets against GPS and GraphLab. Each framework runs four graph algorithms on various graph datasets (as shown in Table 1). Experimental results, as shown in Figure 10, indicate that the speedup of LCC-Graph is higher when running on bigger graph datasets. For example, LCC-Graph is 27.3x, 44.1x and 49.6x faster than GPS respectively when running PR on the LiveJournal, Twitter and UK-2007-05 graphs. The reason for this is the higher scalability of LCC-Graph, since more compute nodes required by bigger graph dataset bring higher speedup. On the contrary, the speedup of GraphLab is lower when running on bigger graph datasets. The reason is that the scalability of GraphLab is slightly lower than GPS. We also notice that the improvement in speed is lower when executing SSSP compared with other graph algorithms (PR, CC and CD). This is because SSSP generates a few interactions among vertices. This restricts the advantage of low communication costs in LCC-Graph.

In our comprehensive evaluation, LCC-Graph runs 12x-49.6x and 6.7x-14.5x faster than GPS and GraphLab respectively on various graph algorithms and graph datasets.

VI. RELATED WORK

Existing BSP-based distributed graph-processing frameworks routinely suffer from high communication costs, which greatly hinder their system performance. To address this problem, the existing studies have focused on the communication efficiency issue of distributed graph-processing frameworks. The *message buffering* technique [2], [3] is used by several BSP-based distributed graph-processing frameworks to amortize the average overhead of each message. This technique



(a) LiveJournal + 3nodes (b) Twitter-2010 + 24nodes (c) UK-2007-05 + 32nodes

Fig. 10. LCC-Graph vs. GPS & GraphLab.

can improve the communication efficiency by sending the message batches, but the improvement is limited, as mentioned in Section II.

Pregel [2] adopts a combiner to reduce the number of cross-machine messages. GiraphUC [22] adopts a barrierless asynchronous parallel (BAP) computation model to reduce both message staleness and global synchronization. However, due to poor spatial locality among the destination vertices, only a relatively small number of messages can be combined. Furthermore, these solutions introduce extra overheads. Finally, a combiner may not be useful in many graph algorithms where the values of the messages are not commutative or combinative. Another alternative solution to reduce cross-machine messages is using advanced graph partitioning strategies [3], [19], [23], [24] to lower the number of cut-edges across nodes. These strategies are also useful for LCC-Graph.

Several systems, such as GraphX [25] and PowerGraph [19], can reduce communication cost by partitioning vertices instead of edges among subgraphs to evenly distribute edges of high-degree vertices, but they also incur high communication cost among partitioned low-degree vertices. However, LCC-Graph reduces the communication cost by eliminating the high extra volume of communication, avoiding the data copying and batch parsing overheads, and by improving the communication bandwidth utilization.

VII. CONCLUSION

This paper proposes a distributed graph-processing framework, called LCC-Graph, to support large-scale graph-computing jobs. LCC-Graph is high-performance and highly scalable while maintaining the advantages of Pregel-like distributed graph-processing frameworks. In LCC-Graph, the LCC-BSP computation model is proposed to eliminate the high communication costs that affect existing distributed graph-processing frameworks, and reduce the computation workload of each vertex. Extensive prototype evaluation of LCC-Graph, driven by real-world datasets, indicates that the performance of LCC-Graph is notably superior to the existing distributed graph-processing frameworks. For example, it runs ∼49x and ∼14x faster than GPS and GraphLab respectively.

ACKNOWLEDGMENT

This work is supported in part by the National High Technology Research and Development Program (863 Program) of China under Grant No.2013AA013203, National Basic Research 973 Program of China under Grant 2011CB302301, and NSFC61173043. This work is also supported by Key Laboratory of Information Storage System, Ministry of Education and State Key Laboratory of Computer Architecture (No.CARCH201505).

REFERENCES

[1] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[2] G. Malewicz, M. H. Austern, and etc., "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD'10*.

[3] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proc. ACM SSDBM'13*.

[4] "Apache giraph." http://giraph.apache.org.

[5] "Goldenorb." http://www.raveldata.com/goldenorb/.

[6] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy." in *CIDR*, 2013.

[7] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Euro-Par 2014 Parallel Processing*. Springer, 2014, pp. 451–462.

[8] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc." in *OSDI'12*.

[9] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[10] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[11] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," Technical Report CMU-CALD-02-107, Carnegie Mellon University, Tech. Rep., 2002.

[12] L. Di Stefano and A. Bulgarelli, "A simple and efficient connected components labeling algorithm," in *Image Analysis and Processing. Proceedings. International Conference on*. IEEE, 1999, pp. 322–327.

[13] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.

[14] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. ACM SIGKDD'12*.

[15] N. Xu, L. Chen, and B. Cui, "Loggp: a log-based dynamic graph partitioning method," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1917–1928, 2014.

[16] I. T. Association *et al.*, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[17] Y. Low, D. Bickson, and etc., "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proc. VLDB'12*.

[18] "Graphlab." http://graphlab.org.

[19] J. E. Gonzalez, Y. Low, and etc., "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI'12*.

[20] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proc. ACM SIGKDD'06*.

[21] Web Algorithmics Lab. http://law.di.unimi.it/datasets.php.

[22] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proc. VLDB'15*.

[23] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *Proc. ACM SOSP'13*.

[24] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proc. ACM SIGMOD'12*.

[25] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: Unifying data-parallel and graph-parallel analytics," *arXiv preprint arXiv:1402.2394*, 2014.